CSC263 Assignment #2

You are allowed to work in a group of at most 3 members, and make one submission as a group. You should clearly write names of all group members in the first page of your submission. Everyone in the same group will get the same marks.
Please read and understand the policy on Academic Honesty on the course website. Then, to protect yourself, list on the front of your submission every source of information you used to complete this homework (other than your own lecture and tutorial notes). For example, indicate clearly the name of every student with whom you had discussions, the title and sections of every textbook you consulted (including the course textbook), the source of every web document you used (including documents from the course webpage), etc.
For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks will be deducted for incorrect or ambiguous use of notation and terminology, and for making incorrect, unjustified, ambiguous, or vague claims in your solutions.

1. (*10 points*) AVL Insert/Delete.

(a) (*4 points*) Write pseudocode for a procedure Balance(*root*, *x*), which takes a subtree rooted at *x* whose left and right subtrees are balanced (subtrees rooted at *x.left* and *x.right* are both AVL trees), and have heights that differ by at most 2, i.e., $|x.right.height - x.left.height| \leq 2$, and alters the subtree rooted at *x* to be an AVL tree.

As given in class, the attributes of each node *x* includes: *key, parent, left, right, height, bf* (balance factor). You are allowed to use the procedures we defined in class directly, such as RightRotate(*root*, *x*), LeftRotate(*root*, *x*), etc. Do not fix the balanceness of *x.parent* or other ancestors yet; you will do this in the next questions.

(b) (*3 points*) Write the pseudocode for AVLInsert(*root*, *x*) to insert a new node *x* to an AVL tree rooted at *root*.

(c) (*3 points*) Write the pseudocode for AVLDelete(*root*, *x*) to delete a node *x* in an AVL tree rooted at *root*.

———————

(a)

```
Balance(root, x)
        if x.bf == −2                                   // right rotate or left-right rotate
                if x.left.bf == 1
                        root = LeftRotate(root, x.left)
                root = RightRotate(root, x)
        else if x.bf == 2                               // left rotate or right-left rotate
                if x.left.bf == -1
                        root = RightRotate(root, x.right)
                root = LeftRotate(root, x)
        return root
```

(b)

```
RetraceBalance(root, x)
        while x ≠ NIL                                   // retrace back to root to update height and bf
                update x.height and x.bf
                y = x.parent
                if |x.bf| = 2
                        root = Balance(root, x)
                x = y
        return root

AVLInsert(root, z)
        x = root
        y = NIL
        while x ≠ NIL
                y = x
                if z.key < x.key
                        x = x.left
                else
                        x = x.right
        z.parent = y
        if y = NIL
                root = z
        else if z.key < y.key
                y.left = z
        else
                y.right = z
                                                // above is the same as Insert of BST
        root = RetraceBalance(root, z)          // retrace back to root to update height and bf
        return root

(c)
AVLDelete(root, z)
        x = z.parent
        if z.left == NIL
                Transplant(root, z, z.right)
        else if z.right == NIL
                Transplant(root, z, z.left)
        else
                y = Minimum(z.right)
                if y.parent ≠ z
                        x = y.parent
                        Transplant(root, y, y.right)        //delete y
                        y.right = z.right
                        y.right.parent = y
```

```
            root = Transplant(root, z, y)
            y.left = z.left
            y.left.parent = y

                                        // above is the same as Delete of BST
    root = RetraceBalance(root, x)      // retrace back to root to update height and bf
    return root
```

_____

2. (*10 points*) Augmenting AVL Tree.

Let $a_1, a_2, ..., a_n$ be a sequence of real numbers, for $n > 1$. (The numbers are in arbitrary order.) A **SeqSet** is an ADT which stores the sequence and supports the following operations:

- PartialSum(S, m): return $a_1 + a_2 + \ldots + a_m$, the partial sum from a_1 to a_m ($1 \le m \le n$).
- Change(S, i, b): change the value of a_i to a real number b.

Design a data structure that implements SeqSet, using an *augmented AVL tree*. The worst-case running time of both PartialSum and Change must be in $O(\log n)$. Describe your design by answering the following questions.

(a) (*2 points*) What is the key of each node in the AVL tree? What other attributes are stored in each node?

(b) (*5 points*) Write the pseudocode of your PartialSum operation, and explain why your code works correctly and why its worst-case running time is $O(\log n)$. Let *S.root* denote the root node of the AVL tree. *Hint: This pseudo-code can be very similar to that of an operation in the textbook, which one is it?*

(c) (*3 points*) Describe in clear English how your Change operation works, and explain why it runs in $O(\log n)$ time while maintaining the attributes stored in the nodes of the AVL tree.

(a)

- *key*: the position of the number in the sequence. For example, the number a_i has key i
- *value*: the value a_i itself
- *sum*: the summation of the numbers in all nodes of the subtree rooted at the current node

```
        For convenience, we define NIL.sum = 0
        x.sum = x.value + x.left.sum + x.right.sum
```

(b)

```
PartialSum'(x, i)
    if x == NIL
        return 0
    if i == x.key
        return x.left.sum + x.value
    if i > x.key
        return x.left.sum + x.value + PartialSum'(x.right, i)
    return PartialSum'(x.left, i)
```

PartialSum(*S*, *i*)
    return PartialSum'(*S.root*, *i*)

We start from the root, going down the tree until we find the node with key *i*. The number of calls of PartialSum' is at most the height of tree $O(\log n)$. The running time is bounded by $O(\log n)$.

(c)

(1) We first search the tree to find the node with key *i*.
(2) Update its *value* to the new value *b*, and update its *sum*.
(3) Then going up the tree to the root, update *sum* of all nodes along the path.

The number of nodes visited is at most the height of tree $O(\log n)$. The *sum* attribute at each node can be updated in constant time. The running time is bounded by $O(\log n)$.

3. (*5 points*) The Division Method of Hashing.

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p.

For an example of interpreting a character string as an integer expressed in suitable radix notation, we might interpret the string "pt" as the pair of decimal integers (112, 116), since "p" = 112 and "t" = 116 in the ASCII character set; then, expressed as a radix-128 integer, "pt" becomes 112×128+116 = 14452.

Prove that if we can derive string x from string y by permuting its characters, then x and y hash to the same value.

We have the following equation:

$$(ab) \bmod c = [(a \bmod c)(b \bmod c)] \bmod c.$$
$$(a + b) \bmod c = [(a \bmod c) + (b \bmod c)] \bmod c.$$

It is easy to see that

$$2^p \bmod (2^p - 1) = 1$$
$$2^{kp} \bmod (2^p - 1) = 1$$

For any a and k, we have

$$a2^{kp} \bmod (2^p - 1) = [(a \bmod (2^p - 1))(2^{kp} \bmod (2^p - 1))] \bmod (2^p - 1)$$
$$= [a \bmod (2^p - 1)] \bmod (2^p - 1)$$
$$= a \bmod (2^p - 1)$$

Therefore, for any string $x = \langle a_{n-1}a_{n-2}\ldots a_1a_0 \rangle$ of length n, it radix 2^p integer representation is

$$\underline{x} = a_{n-1}2^{(n-1)p} + a_{n-2}2^{(n-2)p} + \ldots\ldots + a_1 2^p + a_0$$
$$\underline{x} \bmod (2^p - 1) = (a_{n-1} + a_{n-2} + \ldots\ldots + a_1 + a_0) \bmod (2^p - 1)$$

Thus by permuting the characters, we will not change this value.

4. (*5 points*) Deletion in Open-Address Hashing.

As described in class, when we delete a key from a slot in an open-address hash table, we simply mark the slot by storing the special value DELETED (instead of NIL). This question asks you to implement this idea.

(a) (*4 points*) Write the pseudocode of HashSearch(T, k) and HashInsert(T, k) by modifying the ones we gave in class.

(b) (*1 point*) Write pseudocode for HashDelete(T, k) to delete key k from hash table T. (You need to call HashSearch to find the slot where k is stored.)

(a)

```
HashSearch(T, k)
        for i = 0 to m − 1
                j = h(k, i)
                if T[j] == k
                        return j
                if T[j] == NIL
                        return NIL
                if T[j] == DELETED              // this two added lines are actually unnecessary
                        continue
        return NIL

HashInsert(T, k)
        for i = 0 to m − 1
                j = h(k, i)
                if T[j] == NIL or T[j] == DELETED
                        T[j] = k
                        return j
        error "hashtable overflow"

HashDelete(T, k)
        j = HashSearch(T, k)
        if j = NIL
                return
        T[j] = DELETED
```

5. (*10 points*) Longest-Probe Bound for Hashing

Suppose that we use an open-addressed hash table of size m to store $n \le m/2$ items.

(a) (*2 points*) Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes.

(b) (*3 points*) Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2\log(n)$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\mathbf{Pr}[\, X_i > 2\log(n)\,] = O(1/n^2)$. Let the random variable $X = \max_{1 \le i \le n}\{X_i\}$ denote the maximum number of probes required by any of the n insertions.

(c) (*3 points*) Show that $\mathbf{Pr}[\, X > 2\log(n)\,] = O(1/n)$.

(d) (*2 points*) Show that the expected length $\mathbf{E}[X]$ of the longest probe sequence is $O(\log n)$.

(a)
The load factor before the ith insertion is $(i - 1/m)$.
The probability that the ith insertion requires more than k probes is

$$[(i-1)/m] * [(i-2)/(m-1)] * \ldots\ldots * [(i-k)/(m-k+1)$$
$$\le \quad (n/m)^k$$
$$\le \quad (1/2)^k$$

(b)
For $k = 2\log(n)$, the probability in (a) is at most $(1/2)^{2\log n} = 1/n^2$

(c)
By the fact that $\mathbf{Pr}[\text{ A or B }] \le \mathbf{Pr}[\text{ A }] + \mathbf{Pr}[\text{ B }]$,

$$\mathbf{Pr}[\, X > 2\log(n)\,] \le \mathbf{Pr}[\, X_1 > 2\log(n)\,] + \ldots\ldots + \mathbf{Pr}[\, X_n > 2\log(n)\,]$$
$$\le \quad n * (1/n^2)$$
$$= \quad 1/n$$

(d)
Let $k = \lceil 2\log(n) \rceil$. Then,

$$\mathbf{E}[\, X\,] = \sum_{i=1\ldots n}(i * \mathbf{Pr}[\, X = i\,])$$
$$= \sum_{i=1\ldots k}(i * \mathbf{Pr}[\, X = i\,]) + \sum_{i=k+1\ldots n}(i * \mathbf{Pr}[\, X = i\,])$$
$$\le \sum_{i=1\ldots k}(k * \mathbf{Pr}[\, X = i\,]) + \sum_{i=k+1\ldots n}(n * \mathbf{Pr}[\, X = i\,])$$
$$\le k * (\sum_{i=1\ldots k}\mathbf{Pr}[\, X = i\,]) + n * (\sum_{i=k+1\ldots n}\mathbf{Pr}[\, X = i\,])$$
$$\le k * 1 + n * \mathbf{Pr}[\, X > k\,]$$
$$\le k + n * (1/n)$$
$$\le k + 1$$
$$= O(\log n)$$

6. (*10 points*) Find *k*-th Element

The problem of finding the k-th element is that, given an unsorted array to $A[1\ldots n]$, find the k-th element when all the elements are sorted. The problem finding the median is a special case of this problem for $k = \lceil n/2 \rceil$.

Consider the following idea on finding the k-th element (called RandSelect):

- Randomly pick an element as the pivot.
- Partition the array around the pivot (same as the partitioning in quick sort, with the pivot in the middle, smaller elements in the left part, and larger elements in the right part).
- Let l be the index of the pivot:
  - If $l = k$, return the pivot;
  - If $l < k$, recurse to find the $(k - l)$-th element in the right part
  - If $l > k$, recurse to find the k-th element in the left part

In this problem, we use indicator random variables to analyze the RandSelect procedure in a manner akin to our analysis of RandQuickSort in class.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as $z_1, z_2, \ldots, z_n$, where z_i is the ith smallest element. Thus, the call RandSelect(A, 1, n, k) returns z_k.

For $1 \le i < j \le n$, let

$X_{ijk} = \mathrm{I}\{z_i$ is compared with z_j some time during the execution of the algorithm to find $z_k\}$.

(a) (*2 points*) Give an exact expression for $\mathbf{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of i, j, and k.)

(b) (*2 points*) Let Y_k denote the total number of comparisons between elements of array A when finding z_k. Show that

$$\mathbf{E}[Y_k] \le 2\left(\sum_{i=1\ldots k}\sum_{j=k\ldots n}[1/(j-i+1)] + \sum_{j=k+1\ldots n}[(j-k-1)/(j-k+1)] + \sum_{i=1\ldots k-2}[(k-i-1)/(k-i+1)]\right)$$

(c) (*2 points*) Show that $\mathbf{E}[Y_k] \le 4n$.

(d) (*2 points*) Conclude that, assuming all elements of array A are distinct, RandSelect runs in expected time $O(n)$.

(e) (*2 points*) What is the worst case running time for RandSelect. Justify your answer.

(a)

If $k < i < j$, then $z_i z_j$ are compare iff the first pivot chosen among $\{z_k \ldots z_i \ldots z_j\}$ is either z_i or z_j

$$\mathbf{E}[X_{ijk}] = 2/(j-k+1)$$

If $i \le k \le j$, then $z_i z_j$ are compare iff the first pivot chosen among $\{z_i \ldots z_k \ldots z_j\}$ is either z_i or z_j

$$\mathbf{E}[X_{ijk}] = 2/(j-i+1)$$

If $i < j < k$, then $z_i z_j$ are compare iff the first pivot chosen among $\{z_i \ldots z_j \ldots z_k\}$ is either z_i or z_j

$$\mathbf{E}[X_{ijk}] = 2/(k-i+1)$$

(b)

We can get the result by summing all cases together for a fixed k.

(1) $k < i < j$: for $j = k + 1, \ldots, n$ and $i = k + 1, \ldots, j - 1$, the sum of all $\mathbf{E}[\, X_{ijk}\,]$ is

$\sum_{j=k+1\ldots n} \sum_{i=k+1\ldots j-1} [2 / \ (j - k + 1)]$ = $\sum_{j=k+1\ldots n} [2(j - k - 1) / \ (j - k + 1)]$

(2) $i \le k \le j$: for $i = 1, \ldots, k$ and $j = k, \ldots, n$, the sum of all $\mathbf{E}[\, X_{ijk}\,]$ is

$\sum_{i=1\ldots k} \sum_{j=k\ldots n} [2 / (j - i + 1)]$

(3) $i < j < k$: for $i = 1, \ldots, k - 2$ and $j = i + 1, \ldots, k - 1$, the sum of all $\mathbf{E}[\, X_{ijk}\,]$ is

$\sum_{i=1\ldots k-2} \sum_{j=i+1\ldots k-1} [2 / \ (k - i + 1)]$ = $\sum_{i=1\ldots k-2} [2(k - i - 1) / \ (k - i + 1)]$

Summing all cases above gives the result for $\mathbf{E}[\, Y_k\,]$.

(c)

(1) Since $(j - k - 1) / \ (j - k + 1) < 1$,

$\sum_{j=k+1\ldots n} [2(j - k - 1) / \ (j - k + 1)]$ $\le$ $2(n - k)$

(2) Let $m = j - i + 1 \ge \ k - i + 1 \ge k$, we have

$\sum_{i=1\ldots k} \sum_{j=k\ldots n} [2 / (j - i + 1)]$ $\le$ $\sum_{i=1\ldots k} \sum_{m=k-i+1\ldots n-i+1} [2 / m]$

$\le$ $\sum_{i=1\ldots k} \sum_{m=1\ldots n} [2 / k]$ (since $m \ge k$)

$\le$ $\sum_{i=1\ldots k} [2\ n / k]$

$\le$ $2n$

(3) Since $(k - i - 1) / \ (k - i + 1) < 1$,

$\sum_{i=1\ldots k-2} [2(k - i - 1) / \ (k - i + 1)]$ $\le$ $2(k - 2)$

The result is direct by summing above cases.

(d)

The algorithm makes at most n calls of Partition, and the running time of all Partition is bounded by $O(Y_k)$. Therefore the total running time is $O(n + Y_k)$, and its expectation is $O(n)$.

(e)

The worst case is $O(n^2)$. It happens when the array is sorted, $k = 1$, and for each call of Partition, the pivot is chosen to be the largest in the subarray passed to Partition. The running time for this case is

$c[n + (n - 1) + \ldots + 1] \ = \ O(n^2)$